DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Primer Trabajo Práctico

14 de Abril de 2010 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Bianchi, Mariano	92/08	bianchi-mariano@hotmail.com
Brusco, Pablo	527/08	pablo.brusco@gmail.com
Di Pietro, Carlos Augusto Lyon	126/08	cdipietro@dc.uba.ar

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Ejercicio 1

1.1. Introducción

El primer problema del presente trabajo consistió en la implementación de un algoritmo capaz de dar solución a la ecuación

$$b^n \ mod \ (n) \tag{1}$$

haciendo uso de alguna de las técnicas algorítmicas aprendidas hasta el momento en la materia. Asímismo, la consigna dictaba que la complejidad final del algoritmo debería ser menor a $O(n)$.

En pos de cumplimentar lo pedido se decidió usar la técnica de *Dividir & Conquistar*[1] para desarrollar el algoritmo. Esta técnica se caracteriza principalmente en dividir la instancia de un problema en instancias más pequeñas, atacar cada una de ellas por separado y resolverlas, para finalmente juntar sus resultados y así producir el resultado final.

1.2. Explicación

La primera solución que se piensa casi de manera intuitiva es la de mutliplicar n veces el número b y luego hallar el resto de dividir ese resultado por n.

$$\underbrace{b.b.b \ldots b.b.b}_{n \ veces} \ mod \ (n) = b^n \ mod \ (n) \tag{2}$$

Sin embargo, la complejidad ese algoritmo es $O(n)$, ya que se realizan n multipicaciones y 1 división, razón por lo cual no cumple con lo pedido en la consigna. Además, puesto que ese algoritmo realizar sucesivas multipliciones de b, cabe la posibilidad de que el valor en que se va acumulando el resultado sobrepase el máximo número entero representable en la computadora produciedo así un overflow y obteniéndose en consecuencia un resultado incorrecto.

Analizando la falla del algoritmo anterior, se observa que lo que se debe evitar es calcular directamente el resultado de b^n ya que ese número podría ser excesivamente grande.
Con esto en mente veamos una manera más conveniente de abordar el problema. Es claro, por definición de congruencia, que resover la ecuación en (1) equivale a hallar el x tal que:

$$b^n \ \equiv \ x \ (n) \ , \quad con \ 0 \ \leq \ x \ < \ n \tag{3}$$

Luego, asumamos por un momento que podemos tener gratis un k tal que:

$$b^{n/2} \ \equiv \ k \ (n) \ , \quad con \ 0 \ \leq \ k \ < \ n \tag{4}$$

Entonces, aplicando el resultado de (4) en (3) obtenemos que:

[1] Poner alguna referencia en donde se explique esta técnica

- Si n es par:

$$\begin{aligned} b^n \equiv x\ (n)\ , \quad con\ 0 \leq x < n \\ b^{n/2}.b^{n/2} &\equiv x\ (n) \\ k.k &\equiv x\ (n) \\ k^2 &\equiv x\ (n) \end{aligned} \tag{5}$$

- Si n es impar par:

$$\begin{aligned} b^n \equiv x\ (n)\ , \quad con\ 0 \leq x < n \\ b.b^{n/2}.b^{n/2} &\equiv x\ (n) \\ b.k.k &\equiv x\ (n) \\ b.k^2 &\equiv x\ (n) \end{aligned} \tag{6}$$

Sin lugar a dudas, las expresiones resultantes en ambos casos son fáciles de resolver ya que $k < n$; al tiempo que también son números mucho menores que b^n.

Volviendo un poco sobre nuestro pasos, se dijo anteriormente en (4) que podíamos asumir que obtener k no tenía un costo computacional significativo. No obstante, esto no siempre es cierto puesto que para obtener k es necesario calcular previamente $b^{n/2}$ el cual puede ser un número nada despreciable en lo que respecta a tamaño en memoria.
Pero esto no representa, dificultad alguna para calcular $b^{n/2}$ puesto que podemos aplicar repetidas veces el procedimiento descripto con anterioridad dividiendo el exponente de b hasta que valga 1, y luego reagrupando los resultados y tomando módulo n hasta obtener finalmente la solución del problema.

1.3. Análisis de la complejidad del algoritmo

A continuación presentamos la complejidad del algoritmo *potenciacion* el cual permite resolver la ecuacion (1).

AAAHHHHHHHHHHHHHH!!!! AYUDA!!!!!!!!!!

potenciacion(base : $\mathbb{Z}$, exp: $\mathbb{Z}$, m: $\mathbb{Z}$)

```
1  Complejidad: T(n)
2  var res : ℤ
3  base ← base mod m ;                                          // O(log2(base))
4  if (base == 0);                                              // O(log2(base))
5  then
6  |  return base
7  else
8  |  if (exp == 1);                                            // O(log2(exp))
9  |  then
10 |  |  return base
11 |  else
12 |  |  if (exp mod 2 == 0) ;                                  // O(log2(exp))
13 |  |  then
14 |  |  |  var temp : ℤ
15 |  |  |  temp ← potenciacion(b, exp/2, m) ;                      // T(exp/2)
16 |  |  |  temp ← (temp*temp) ;                          // O(log2(log2(temp)^2))
17 |  |  |  temp ← temp mod m ;                  // O(log2(log2(temp) * log2(m)))
18 |  |  |  res ← temp ;                                    // O(log2(temp))
19 |  |  else
20 |  |  |  var temp : ℤ
21 |  |  |  temp ← potenciacion(b, (exp-1)/2, m) ;              // T((exp-1)/2)
22 |  |  |  temp ← (temp*temp) ;                          // O(log2(log2(temp)^2))
23 |  |  |  temp ← temp mod m ;                  // O(log2(log2(temp) * log2(m)))
24 |  |  |  temp ← temp * base ;                  // O(log2(log2(temp) * log2(b)))
25 |  |  |  temp ← temp mod m ;                  // O(log2(log2(temp) * log2(m)))
26 |  |  |  res ← temp ;                                    // O(log2(temp))
27 |  |  end
28 |  end
29 end
30 return res
```

1.4. Detalles de Implementación

Dentro de la carpeta *./ej1/*, se puede encontrar un archivo ejecutable *ejercicio_1* compilado para GNU-Linux, el cual resuelve el problema anteriormente descripto. Este programa se ejecuta por consola mediante el comando `./ejercicio_1`, y recibe como parámetro los archivos de entrada *".in"* a procesar. Puede recibir tantos nombres de archivo como se desee, pero en caso de no recibir ninguno, el programa procesará el archivo *Tp1Ej1.in* que se encuentra incluído dentro de la misma carpeta.
Una vez ejecutado, el algoritmo procesa la cola de archivos que recibió como parámertos de a uno por vez generando para cada uno de ellos dos archivos:

- Un archivo *".out"* omónimo con la solucion a la ecuación (1) para cada par de naturales (b, n) contenidos en el archivo de entrada. Este archivo se guarda en la misma carpeta

en la que se encuentra el ejecutable *ejercicio_1*

- Un archivo omónimo con el sufijo *"_grafico.out"*, en el cual registra para cada n el número de operaciones realizadas por el algoritmo. Este archivo se guarda en la carpeta *./ej1/info graficos*, y tiene por objetivo facilitar la tarea de cargar los datos en el programa de análisis gráfico *QtiPlot*.

Por otra parte, en la misma carpeta, se puede hallar un archivo ejecutable *input_gen1* también compilado para GNU-Linux. Al correr este programa desde la consola mediante el comando `./input_gen1` se despliega un menú de opciones para generar distintos tipos de archivos *".in"* para ser resueltos por el ejecutable *ejercicio_1*. Una vez elegido el tipo de entrada a crear, el programa solicita se que ingrese un nombre de archivo y la cantidad de casos a generar. Acto seguido guarda el archivo generado en la carpeta *./ej1/*.

Asímismo, en la carpeta *./ej1/* encontramos un Makefile el cual permite recompilar los archivos ejecutables con tan solo ejecutar el comando `make` en la consola. Además, ejecutando el comando `make clean` podemos eliminar los archivos ejecutables y todos los archivos de extensión *.out*.

Luego, en la carpeta *./ej1/sources* se encuentran los codigos fuentes de los ejecutables antes descriptos. Los mismo están escritos en lenguaje C++ y tienen comentadas las partes relevantes para simplificar la comprensión.

Finalmente, en *./ej1/* encontramos los archivos *.Tp1Ej1.in* y *Tp1Ej1.out* que vienen con junto con en el enunciado de este Trabajo Práctico, y encontramos también la carpeta *./ej1/test* en donde se hallan los archivos *".in"* generados para probar el algortimo que resuelve (1) junto con sus correspondientes archivos *".out"*, *"_graficos.out"* y los gráficos de cantidad de operaciones vs. n.

2. Ejercicio 2

2.1. Introducción

El segundo problema consistió en la implementación de un algoritmo capaz de dar solución al problema que se plantea a continuación:

Decidir si un grupo de n personas pueden formar o no una ronda que cumpla con las siguientes restricciones:

- La ronda debe contener a todas las personas.
- Algunas personas son amigas y otras no.
- Cada alumna debe tomar de la mano a dos de sus amigas.

2.2. Explicación

Dado que para este problema no se conocen algoritmos buenos[2], se pensó en utilizar la solución por fuerza bruta, es decir, intentar todas las combinaciones hasta lograr determinar si hay o no solución, pero, agregando algunas mejoras.

Mejoras como evaluar, en el momento de cargar la ronda, si alguna persona no tiene las suficientes amigos, es decir 2, o si todos son amigos de todos.

También a este método, se le agregó la estrategia de "Vuelta atrás", (Backtracking).

Backtracking:
En su forma básica, la idea de backtracking se asemeja a un recorrido en profundidad dentro de un grafo dirigido. El grafo en cuestión suele ser un árbol, o por lo menos no contiene ciclos. Sea cual sea su estructura, existe sólo implícitamente. El objetivo del recorrido es encontrar soluciones para algún problema. Esto se consigue construyendo soluciones parciales a medida que progresa el recorrido; estas soluciones parciales limitan las regiones en las que se puede encontrar una solución completa. El recorrido tiene éxito si, procediendo de esta forma, se puede definir por completo una solución. En este caso el algoritmo puede bien detenerse (si lo único que se necesita es una solución del problema) o bien seguir buscando soluciones alternativas (si deseamos examinarlas todas). Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar. En tal caso, el recorrido vuelve atrás exactamente igual que en un recorrido en profundidad, eliminando sobre la marcha los elementos que se hubieran añadido en cada fase. Cuando vuelve a un nodo que tiene uno o más vecinos sin explorar, prosigue el recorrido de una solución. [3]

En este caso particular, la idea es, seleccionar una persona (P), ingresarla en la ronda,

[2] Un algoritmo se considera bueno si puede ser resuelto en tiempo polinomial.
[3] http:es.wikipedia.org

3. Ejercicio 3

3.1. Introducción

El tercer y último problema de este trabajo práctico consiste en, dadas dos listas con horarios de entrada y salida de ciertos trabajadores a una empresa, decidir cuál es el mayor número de los mismos que se encuentran al mismo tiempo dentro de dicha empresa.

En una primer mirada, se pensó que la mejor forma de resolver el ejercicio era aplicando un algoritmo similar al del quicksort[4], sólo que el mismo se realizaba sobre conjuntos y no sobre arreglos. En resumidas cuentas, la idea era tomar un trabajador cualquiera del grupo, y separar en tres conjuntos distintos: los que se cruzaban con el trabajador pivote, los que salían antes que el pivote entrara y los que entraban luego que el pivote saliera[5]. De esta manera, si se repetía el proceso varias veces, se llegaba a tener varios conjuntos en los cuales aparecían solamente trabajadores que se cruzaban en sus horarios dentro de la empresa. Finalmente, sólo restaba devolver el mayor cardinal de dichos conjuntos.

Al igual que el quicksort, si el pivote era elegido al azar, el algoritmo anterior contaba con una complejidad promedio de $O(n * log(n))$ (donde n es la cantidad de trabajadores). Igualmente, el peor caso seguía siendo $O(n^2)$, por lo que no se iba a poder respetar la cota dada como máxima para este trabajo, ya que se especificaba que el algoritmo debía tener una complejidad menor a $O(n^2)$.

Pero luego de un mejor análisis del problema, se cayó en la cuenta que los datos de entrada contaban con la característica de estar ordenados por horario (tanto los de entrada como los de salida), por lo que inmediatamente surgió la idea de poder solucionar el problema con una complejidad de $O(n)$.

Para lograr dicha solución, no se requirió de una técnica compleja. En realidad, lo que se hizo fue iterar sobre las dos listas de entrada, que contenían los horarios de entrada y de salida de los trabajadores. De esta forma, con un recorrido lineal sobre ambas listas de entrada, se podía saber con certeza cuál era la respuesta al problema para las mismas.

3.2. Explicación

Para encontrar solución al problema dado, lo primero que se realizó fue pensar qué cosas se necesitaban para representarlo. Para ello se creó una clase "Empresa", la cual consta de dos arreglos que contienen en cada posición, una hora junto con un nombre de un trabajador, ambos arreglos ordenados ascendentemente por hora. Uno de ellos contiene los horarios de entrada de los trabajadores, mientras que el otro contiene los de salida. Además, se cuenta con las siguientes precondiciones:

- Cada trabajador ingresa estrictamente antes de egresar

[4] Alguna referencia que explique el algoritmo

[5] Para que el algoritmo funcionara bien, se debía tener cierto cuidado con los trabajadores que se cruzaran con el pivote, pero la explicación de estos detalles no hacen a la escencia de la introducción. Además, este algoritmo fue descartado más adelante, por lo que estos detalles son irrelevantes para este trabajo.

- Los horarios van desde 00:00:00 hasta 23:59:59.

Una vez cargados los datos en nuestra estructura, se procedió a armar el algoritmo que resuelve el problema. La idea del mismo fue ir indexándo ambos arreglos desde la posición 0 hasta la n-1 ésima. De esta forma, lo que se hizo fue ir recorriendo el arreglo que contenía los horarios de entrada, hasta que se encontrara una posición en la cuál su horario de entrada fuese mayor al horario de salida que se estaba indexando en ese momento (en el caso de la primer iteración por ejemplo, el horario ubicado en la posición 0 del arreglo de horarios de salida). Mientras se realizaba esto, un contador con valor inicial 0 se iba incrementando con cada iteración, simulando así la cantidad de trabajadores que iban entrando.

Cuando se encontraba un horario de entrada mayor al de salida esto significaba que, antes de que ingresara un nuevo trabajador, al menos uno se había retirado, por lo que se salía de esta iteración. En un primer paso se verificaba cuántos trabajadores habían entrado hasta ese momento y se guardaba en una variable (siempre y cuando dicho valor fuese mayor al guardado anteriormente). Luego, se recorría el arreglo que contenía los horarios de salida de la misma manera que antes: recorrer hasta que se encuentre un horario mayor al indexado en el arreglo de entradas, con la diferencia que en este caso, cada vez que se iteraba, se decrementaba el contador de trabajadores simulando un egreso.

Esto se repetía varias veces, hasta que se indexaran todas las posiciones del arreglo de los horarios de entrada, devolviendo finalmente el valor que se iba actualizando tras cada cambio de iteración, es decir, el valor que se actualizaba una vez que se dejaba de iterar el arreglo con los horarios de entrada.

A modo de una explicación más clara que nos acerque un poco más a la implementación, detallamos lo expreso anteriormente a través del siguiente pseudocódigo:

```
1  Complejidad: O(n)
2  var i,j,maxJuntos,juntosPorAhora : ℤ
3  var termine : bool
4  i ← j ← maxJuntos ← juntosPorAhora ← 0;                          // O(1)
5  termine ← false;                                                 // O(1)
6  while (!termine);                                                // O(n)
7  do
8  |  while (noLlegueAlFinal(i,n) && hora(entradas[i]) ≤ hora(salidas[j]));
   |  // O(1)
9  |  do
10 |  |  juntosPorAhora++ ;                                        // O(1)
11 |  |  i++ ;                                                     // O(1)
12 |  end
13 |  termine = i ≥ n ;                                           // O(1)
14 |  if (maxJuntos ≤ juntosPorAhora);                            // O(1)
15 |  then
16 |  |  maxJuntos ← juntosPorAhora;                               // O(1)
17 |  end
18 |  while (!termine && hora(salidas[j]) ≤ hora(entradas[i]));    // O(1)
19 |  do
20 |  |  juntosPorAhora−− ;                                        // O(1)
21 |  |  j++ ;                                                     // O(1)
22 |  end
23 end
24 return maxJuntos
```

3.3. Análisis de la complejidad del algoritmo

Para realizar el análisis de la complejidad del algoritmo, se decidió utilizar el modelo uniforme y no el logarítmico. Esto se debió a que en este caso, no resulta lógico evaluar la complejidad de acuerdo al costo de representar los valores de los parámetros de entrada. Más aún, por la forma y el contexto del problema y por el algoritmo implementado para la resolución, no sería correcto hacer un análisis logarítmico ya que en este caso las horas representables están acotadas (habíamos dicho que se encontraban entre 00:00:00 y 23:59:59) y la cantidad de trabajadores, aunque no está explícitamente acotada, podríamos suponerla así. De esta manera, sería válido considerar que la complejidad espacial de cada elemento es unitaria, haciendo inadecuado analizar la complejidad del algoritmo con el modelo logarítmico.

Con el objetivo de realizar un mejor análisis de la complejidad del algoritmo propuesto, vamos a analizar el mismo remitiéndonos al pseudocódigo citado en la sección [**??**].

Si hacemos una mirada mas minuciosa sobre el pseudocódigo, veremos que la complejidad del algoritmo está dada por n, que hace referencia a la cantidad de trabajadores de la empresa. Si vemos las complejidades de cada operación, vemos que todas son constantes, con excepción del flujo "while" más grande (aquél que tiene como condición: (!*termine*)). Veamos por qué sucede esto.

Como se puede ver en la línea 5, el booleano *termine* es inicializado con el valor de verdad *false*. La otra variable que nos va a interesar para este análisis es i, la cuál se inicializa con el valor 0. Con estos valores, y sabiendo que el flujo while que estamos analizando tiene como condición la negación de *termine*, se desprende que el algoritmo terminara sí y solo sí *termine* tome el valor de verdad *true*.

Si vemos la línea 13 del pseudocódigo, vemos que aquí es modificado el valor de verdad de *termine* y no se modifica en ningún otro lugar. En esta línea se ve que:

$$termine \leftarrow i \geq n \tag{7}$$

Entonces, juntando esto con lo dicho anteriormente, se ve claramente que el algoritmo va a finalizar una vez que $i \geq$ n. Veamos entonces cómo se comporta i a lo largo del programa.

La variable i sólo es modificada en el primer while anidado, y en esta modificación, i es incrementada de a uno por vez. Por lo tanto, una vez que se ejecute el contenido de este while anidado n veces, el algoritmo finalizará.

Si consideramos la precondición de que todos los trabajadores ingresan a la empresa estrictamente antes de egresar, podemos ver entonces que este primer while anidado va a ejecutarse siempre a lo sumo n veces[6]. Por lo tanto, el algoritmo tiene una complejidad de $O(n)$.

3.4. Detalles de Implementación

Para compilar este programa, se debe referir a la carpeta que contiene al ejercicio nº 3 y ejecutar make en consola.

Luego de ejecutar make, se creara un archivo ejecutable, llamado main. Para ejecutarlo, se debe hacer una llamada desde la consola respetando el siguiente formato:
`main [nombreEntrada] [nombreSalida]`.

Los parámetros son opcionales, pero se puede elegir entre no pasar ningún parámetro, o pasar 2. Si no se pasan parámetros, el programa tomara los valores por defecto, esto quiere decir que el archivo de entrada será el designado por la materia.

[6] Ver condición del primer while anidado